ACTIVATING REFACTORINGS FASTER

Emerson Murphy-Hill, Portland State University, emerson@cs.pdx.edu

Problem and Motivation

Refactoring is a practice performed by computer programmers to restructure their code without changing its externally visible behavior. The practice was first named and characterized in 1990 by Opdyke and Johnson [Opdyke90], and was later popularized in Martin Fowler's book in 1999 [Fowler99]. Fowler implored programmers to refactor their code continuously to make it easier to understand, to aid in finding and fixing bugs, and to make room for the addition of new features. Research suggests that many programmers practice refactoring frequently [Murphy06,Weißgerber06].

While refactoring can be beneficial, refactoring by hand can be slow because one must be careful not to change program behavior. Refactoring tools were designed to automate this difficult task. For example, with the speed and accuracy of a compiler, a refactoring tool can change the name of a class and go through an entire program to update every reference to that class. Designers of integrated development environments have recognized the value of refactoring tools, and therefore have built refactoring tools for a variety of languages. And because refactoring is a frequent activity, refactoring tools hold promise to significantly increase programmer productivity.

The problem, however, is that programmers do not use refactoring tools as frequently as they could [Hill08b]. In a survey of 112 Agile 2007 conference attendees, I attempted to determine why programmers do not use refactoring tools. One significant finding was that programmers said that they can refactor faster *without* the aid of a tool. Indeed, among programmers who spend at least 10 hours per week programming and have refactoring tools available at least 90% of the time, 40% reported that they can sometimes refactor faster by hand than with a tool. This runs in direct opposition to one of the stated purposes of refactoring tools -- to help programmers refactor faster.

Are refactoring tools really as slow as programmers believe? How can refactoring tools be improved to fit better with the way programmers refactor? The goal of this research is to answer these questions, answers which I hope will help realize the productivity gains promised by refactoring tools.

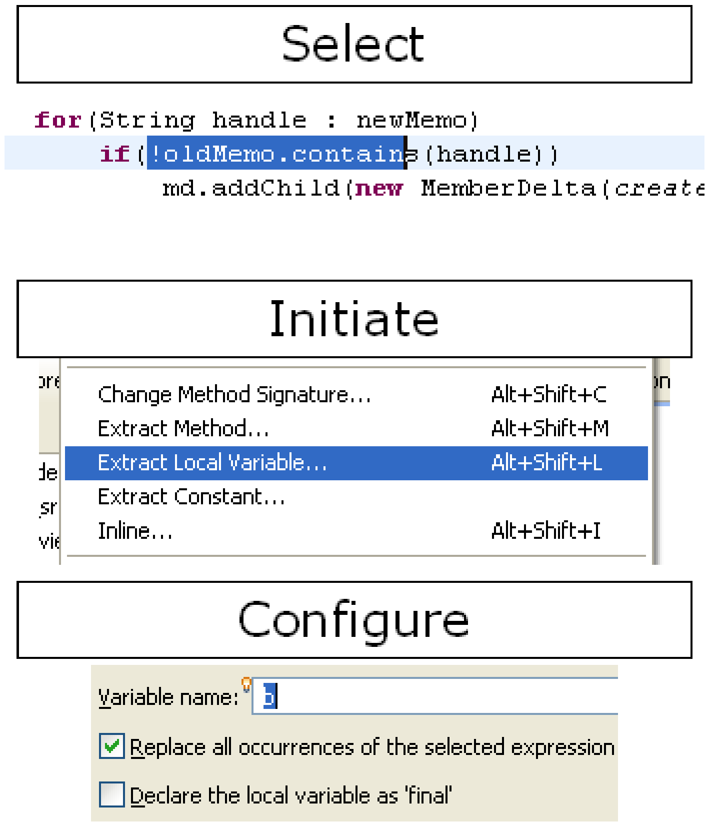
Background & Related Work

Since their introduction over 10 years ago, refactoring tools have been implemented for a variety of languages and programming environments. Refactoring tools have been built for Java in the Eclipse environment [Eclipse], for Objective-C in Xcode [Xcode], for Smalltalk in Squeak [Squeak], and for Visual Basic in Visual Studio [Vstudio], just to name a few. While these tools vary in maturity and the types of refactorings supported, nearly every refactoring tool is identical in that a programmer activates them in three steps. First, the programmer **selects** code to refactor, then **initiates** a specific type of refactoring, and then **configures** the refactoring tool to produce the desired output (Fig. 1). These three refactoring activation steps comprise the core usage of today's refactoring tool.

When describing the design of the first refactoring tool for Smalltalk, Roberts noted that source code transformation speed is critical to the adoption of a refactoring tool [Roberts99]. Likewise, toolbuilders such as the designers of Eclipse have made performance tradeoffs to make refactoring tools acceptably fast [Kiezun07]. Such work differs from this research in that I address refactoring tool speed in the user interface, rather than computationally.

This is not to say, however, that previous work has not attempted to improve the user interface of refactoring tools. In the past, I have shown significant speed improvements to refactoring, but only for one particular refactoring and not for activation or configuration [Hill08a]. Mealy and colleagues have taken a more holistic approach to improving the usability of refactoring tools, but do not suggest any techniques for improving the speed with which programmers use the tools [Mealy07]. The research presented in this paper suggests concrete improvements to the user interfaces of a wide variety of refactoring tools.

Several refactoring tools have improved on the standard mechanisms for selecting, initiating, and configuring a refactoring tool. Some tools, such as Eclipse, allow the programmer to quickly and accurately select code for refactoring by allowing her to select icons representing code entities, such as methods and instance variables [Eclipse]. While this technique allows for efficient program entity selection, it is not applicable for low level refactorings such as Extract Method. Most refactoring tools allow the programmer to initiate a refactoring using a linear menu or hotkey. However, linear menus are relatively slow, especially when a tool must list the many types of refactorings that it supports. Hotkeys are fast, but they are difficult to remember; the programmer has to first think of the restructuring she wants to perform (for example, "take this code and put it into a new temporary variable"), map that onto a capriciously named refactoring ("Extract Local Variable"), and finally map that onto some hotkey (remembering Alt+Shift means "refactoring" and "L" means "Local"). This problem is worsened by the fact that the names of refactorings are non-standard and vary among different programming environments. Most environments allow the programmer to configure a refactoring using a dialog, but such dialogs are typically modal, so the programmer does not have access to her code while refactoring, nor does she typically have access to other programming tools, such as the ability to mouse-over a method call and read its documentation [Eclipse].


Select
Initiate
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Inline... Alt+Shift+I
Configure

```
for(String handle : newMemo)
    if(!oldMemo.contains(handle))
        md.addChild(new MemberDelta(create
```

Variable name: b

[✓] Replace all occurrences of the selected expression

[ ] Declare the local variable as 'final'

Figure 1. The three common steps for using a modern refactoring tool.

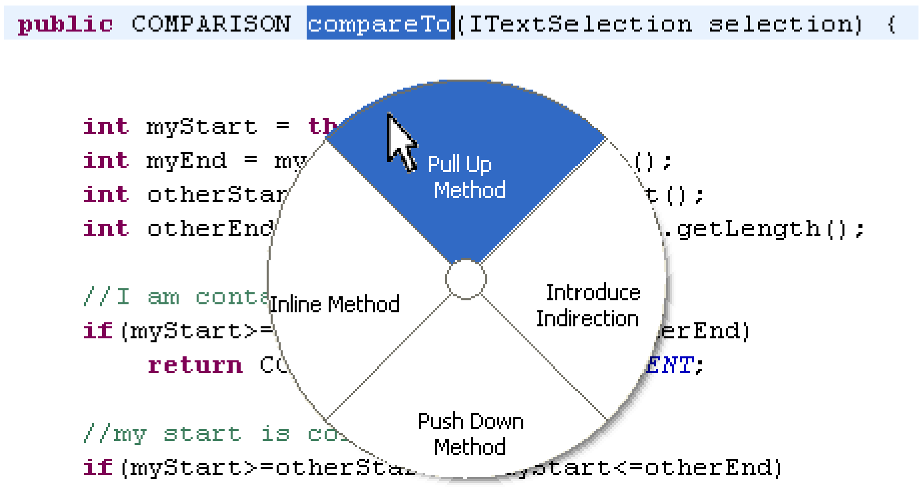
Uniqueness of the Approach

I approach the problem of slow refactoring tools by building two proof-of-concept tools: pie menus for refactoring, which speed up tool initiation, and refactoring cues, which speed up code selection and refactoring configuration. Both tools were built as plugins to the Eclipse environment as extensions to the standard Eclipse refactoring tools. While these tools were designed to address separate steps in the refactoring process, they can be used in conjunction [Hill07].

Pie menus for refactoring are contextual menus that are circular (Fig. 2). Items on the menu can be selected using the mouse, as would traditional linear menus, or can be selected by mouse gesture in the direction of the desired item. Pie menus, also known as marking menus when the user can gesture towards the desired item [Kurtenbach96], were originally designed as a faster alternative to traditional linear menus [Callahan88]. For refactoring, I have limited the number of refactoring items per menu to 4, with no submenus. Currently, a total of 14 refactorings can be initiated using these pie menus in Eclipse.

Using pie menus for refactoring has several advantages. First, as previous research has already shown, pie menus are significantly faster than traditional menus, with no loss in accuracy [Callahan88,Kurtenbach96]. Second, if the user already knows which menu item she wants and where it appears, she doesn't have to wait for menu to be drawn on screen because she can just gesture-ahead in the desired direction [Kurtenbach96]. Third, repeated use of refactoring items reinforces muscle memory, facilitating this gesture-ahead behavior. Fourth, because the number of items on the menu is limited to four, a simple hotkey scheme can be used, making them not only appropriate for users who prefer the mouse, but also for users who prefer the keyboard. Fifth, I claim that pie menus for refactoring are especially memorable because of the chosen *design rationale*, the assignment of refactorings to directions: top means "push into superclass," bottom means "pull down into subclasses," left means "specialize," and right means "generalize." This scheme works well because many refactorings are directional (e.g., Pull

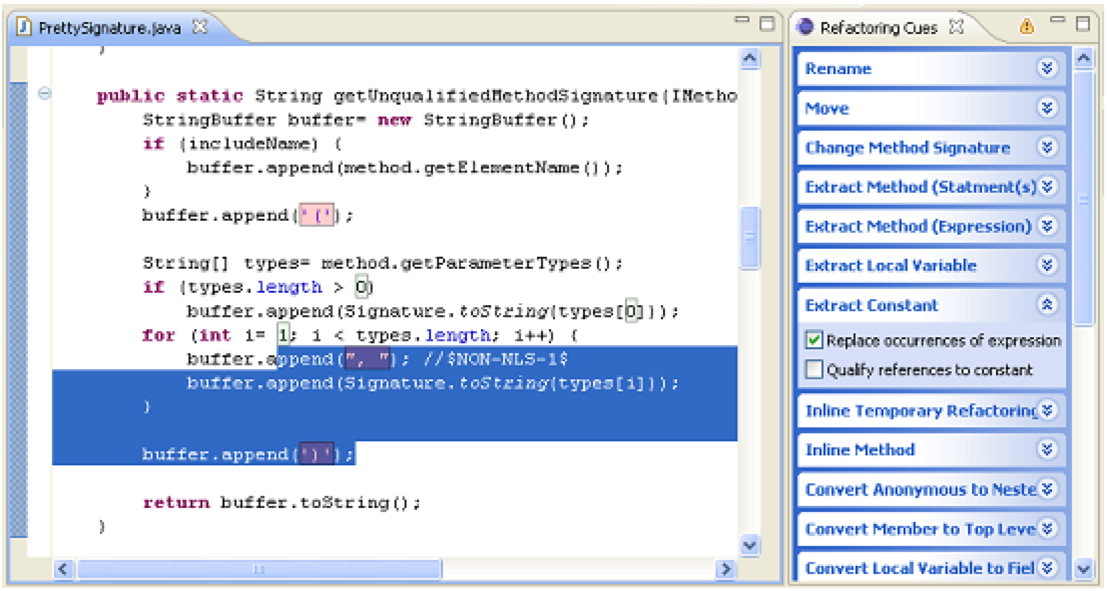
Up Method appears on top), have inverses (Extract Method appears opposite Inline Method), and are conceptually similar (Inline Local Variable and Inline Method both appear on the left). I will discuss a validation of this memorability claim in the next section.



```
public COMPARISON compareTo(ITextSelection selection) {

    int myStart = th
    int myEnd = my                                    ();
    int otherStar                                     t();
    int otherEnd                                      .getLength();

    //I am cont
    if(myStart>=                                      erEnd)
        return CC                                     ENT;

    //my start is co
    if(myStart>=otherSta                 ystart<=otherEnd)
```

Figure 2. A pie menu for refactoring. Refactorings are chosen by keyboard, mouse click, or directional mouse gesture.

There are several limitations of using pie menus over traditional tool activation mechanisms, however. First, while many refactorings seem to fit well onto pie menus, some refactorings, such as the popular Rename refactoring, do not seem to have any natural direction. This problem can be somewhat alleviated by placing such refactorings onto diagonal directions, or by simply requiring users to revert back traditional initiation mechanisms for such refactorings, such as linear menus. Second, adding new items to pie menus will disrupt the users' accumulated muscle memory.



```
    }

    public static String getUnqualifiedMethodSignature(IMetho
        StringBuffer buffer= new StringBuffer();
        if (includeName) {
            buffer.append(method.getElementName());
        }
        buffer.append('(');

        String[] types= method.getParameterTypes();
        if (types.length > 0)
            buffer.append(Signature.toString(types[0]));
        for (int i= 1; i < types.length; i++) {
            buffer.append(", "); //$NON-NLS-1$
            buffer.append(Signature.toString(types[i]));
        }

        buffer.append(')');

        return buffer.toString();
    }
```

PrettySignature.java
Refactoring Cues
Rename
Move
Change Method Signature
Extract Method (Statment(s)
Extract Method (Expression)
Extract Local Variable
Extract Constant
Inline Temporary Refactoring
Inline Method
Convert Anonymous to Neste
Convert Member to Top Leve
Convert Local Variable to Fiel

[✓] Replace occurrences of expression
[ ] Qualify references to constant

Figure 3. Refactoring cues. Program elements are targeted for refactoring by mouse click or cursor selection (those in red). The refactoring can be configured in the palette, at right. Here, two characters and one string is targeted for the Extract Constant refactoring.

Refactoring cues improve the speed at which programmers can select code and configure the refactoring tool. The most notable feature of refactoring cues is that they reverse the traditional select-then-initialize refactoring tool paradigm. First, the user initiates the desired refactoring from a non-modal palette, adjacent to the program editor. The tool then does two things simultaneously: all possible places where the refactoring could apply are overlaid with rectangular cues, and configuration options are displayed in the palette. Next, the user selects cues in the editor as the targets of the refactoring (Fig. 3, left), thereby changing their color from green to red, and can also choose configuration items in the palette (Fig. 3, right). Finally, when the user is satisfied, she presses a button or a hotkey to execute the desired refactorings.

Refactoring cues have several advantages over traditional refactoring tools. First, because the user is selecting program elements in the form of cues rather than individual characters, the user cannot select syntactically invalid input to the refactoring tool. Second, because initiation precedes selection, the user can choose multiple program elements to be refactored at one time. This is a significant use case, as Murphy and colleagues' data show [Murphy06], because nearly half of all refactorings occur in very short periods of time [Hill08c]. Third, because configuration is non-modal, the programmer has access to all tools normally available, and she can avoid looking at the configuration options altogether, if she so desires.

There are several limitations of refactoring cues as well. First, some refactorings can be applied in many places in the code, such as those that apply to expressions, and thus the cues' sometimes overlap and can appear visually complex. Second, because initiation precedes selection, all available refactorings must be displayed in the palette, making the palette quite large. This could be alleviated somewhat by providing search functionality or by grouping related refactorings.

These two tools represent a unique approach to improving the speed of refactoring tools because they focus on the three core activities during refactoring tool activation: selection, initiation, and configuration.

Results and Contributions

To evaluate the two new refactoring user interfaces in terms of usability, I conducted three different evaluations: one for pie menus, one for refactoring cues, and one which asks programmers their opinions of both tools. Full details of these evaluations can be found elsewhere [Hill08c].

Fortunately, previous research has already shown that pie menus are faster than traditional linear menus [Callahan88]. However, I claim that the design rationale with which I have placed refactorings on pie menus is especially memorable, and I provide support for this claim with a memory recall experiment. In this experiment, I recruited 16 subjects: students from a class on design patterns, computer science research assistants, and programmers in industry. During the experiment, I asked half of the subjects to remember the direction of 9 refactorings placed according to the design rationale, and the other half to remember the direction of refactorings placed in a manner *contrary* to the design rationale. Subjects were given 30 seconds to remember the location of each refactoring. Next, subjects were given 5 minutes to recall the location of the same 9 refactorings, albeit in a different order. Additionally, subjects were asked to guess the direction of one additional refactoring that they had not been asked to remember.

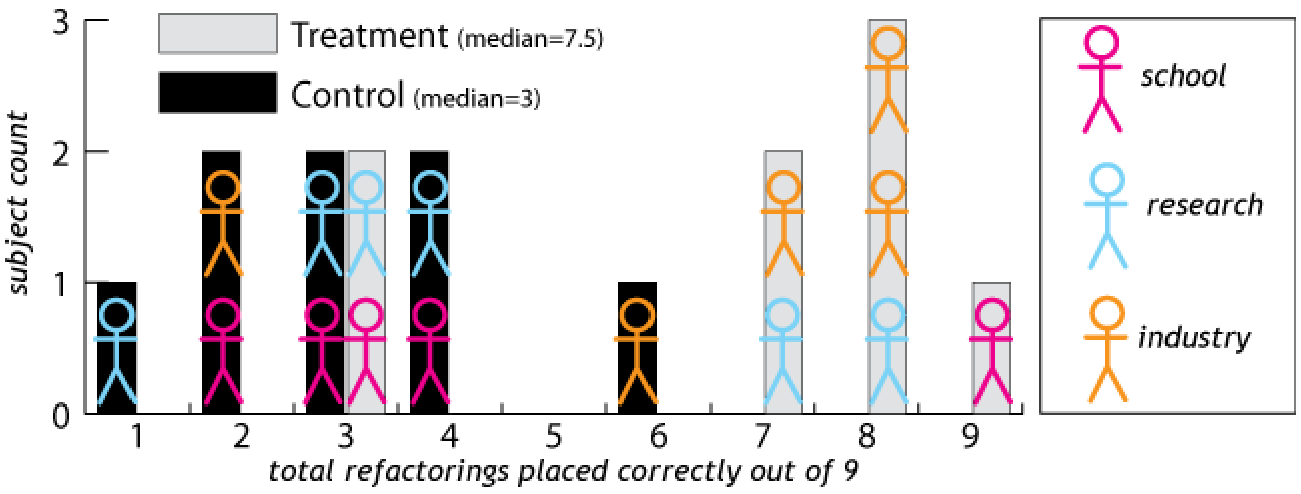


Figure 4. Memory recall experiment results.

Results of the study showed that the programmers who were shown refactorings placed according to the design rationale could recall a median of 4.5 more refactorings than those subjects who were shown refactorings placed against the rationale (Fig. 4), a difference that is statistically significant by a Kruskal-Wallis one-way analysis of variance. Moreover, 6 out of the 8 subjects correctly guessed the location of previously unseen refactoring, suggesting that the subjects were not simply recalling the location of refactorings, but had mentally synthesized a model of the directions in which refactorings *should* appear. The results suggest that (1) refactorings placed according to the design rationale can help programmers activate refactorings more quickly by helping facilitate their transition to the gesture-ahead use of pie menus, and (2) programmers may be able to simply guess the gesture required to perform a refactoring, without having to remember its name. The results of this experiment must be tempered by the fact that the experiment consisted of only 16 subjects and 9 refactorings, which may not be representative of most programmers nor most refactorings.

In the second evalution, to compare refactoring cues and traditional refactoring tools, I compare two models, one of programmers using refactoring cues, and one of programmers using a traditional refactoring tool. The two models compared are NGOMSL models, abstractions of human-computer interaction used for evaluating usability [John96]. The thrust of the argument is that a programmer using refactoring cues can perform exactly the same steps that she would perform using traditional tools, except that the steps are performed in a different order. Therefore, refactoring cues are at least as fast as traditional refactoring tools. Furthermore, in certain cases, such as when several program elements need to be refactored at once, refactoring cues are faster than traditional tools. The interested reader can view the technical details of the evaluation online [Hill08c]. While this evaluation suggests that refactoring cues are at least as fast as traditional refactoring cues, it is limited in that it does not take into account every way that each tool might be used.

I conducted the final evaluation by interviewing programmers regarding their opinions of the tools. The evaluation was conducted by interviewing Java programmers at the 2007 O'Reilly Open Source Convention. I interviewed 15 programmers for about 20 minutes each, the interview consisting of a few background questions, short videos demonstrating the tools used in practice, and some questions regarding the programmers' opinions of the tools. The videos were used both because the tools were prototypes, not yet ready for programmer consumption, and to ensure consistency between interviews. The videos can be viewed as short screencasts at http://multiview.cs.pdx.edu/refactoring/activation

Interviewees responded positively both to pie menus and to refactoring cues. Six subjects said pie menus would encourage them to refactor more frequently, while eight said the same of refactoring cues. Subjects estimated that they were significantly more likely to use pie menus than linear menus for refactoring. Subjects also estimated that they would use refactoring cues, pie menus, and hotkeys all about the same amount. Subjects provided some insight into future directions for the tools as well. For instance, programmers disliked the size and opacity of the menus; I plan to address this concern by implementing a labels-only interface. Programmers were also concerned that the colors used in refactoring cues would not be distinguishable to color-blind people; I plan on investigating how different cues can be distinguished without the need for color. The results suggest that programmers are willing to use pie menus and refactoring cues alongside traditional hotkeys and linear menus.

Refactoring is an import part of the software development process, and refactoring tools can help developers work more efficiently. But first, programmers must be willing to use those tools. This research makes 3 main contributions towards more usable tools:

1. A characterization of how existing refactoring tools slow down programming and data suggesting that programmers believe that speed is a hinderance to refactoring tool usage.
2. Two novel refactoring tool user interfaces designed to increase the speed at which programmers activate refactoring tools.
3. Three studies suggesting that these user interfaces are indeed faster, and that programmers are willing to use them as part of their programming toolset.

References

[Callahan88] Callahan, J., Hopkins, D., Weiser, M., and Shneiderman, B. 1988. An empirical comparison of pie vs. linear menus. In CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, 95--100.

[Eclipse] Eclipse. 2008. The Eclipse Foundation. Computer Program, http://www.eclipse.org.

[Fowler99] Fowler, M. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Hill07] Murphy-Hill, E. and Black, A. P. 2007. High velocity refactorings in Eclipse. In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange. ACM, New York, NY, 1--5.

[Hill08a] Murphy-Hill, E. and Black, A. P. 2008. Breaking the barriers to successful refactoring: Observations and tools for extract method. In Proceedings, International Conference on Software Engineering. IEEE Computer Society, Leipzig, Germany. In Press.

[Hill08b] Murphy-Hill, E. and Black, A. P. 2008. Refactoring tools: fitness for use. IEEE Software. 25, 5. In Press.

[Hill08c] Murphy-Hill, E. and Black, A. P. 2008. Refactoring tools that stay out of the programmer's way. http://amstel.cs.pdx.edu/Members/emerson/TOCHI.pdf . In Preparation.

[John96] John, B. E. and Kieras, D. E. 1996. The GOMS family of user interface analysis techniques: Comparison and contrast. ACM Transactions on Computer-Human Interaction 3, 4, 320--351.

[Kiezun07] Kiezun, A., Fuhrer, R., Keller, K., Advanced Refactoring in Eclipse: Past, Present, and Future. Presentation from 1st Workshop on Refactoring Tools. https://netfiles.uiuc.edu/dig/RefactoringWorkshop/Presentations/AdvancedRefactoringInEclipse.pdf

[Kurtenbach93] Kurtenbach, G. and Buxton, W. 1993. The limits of expert performance using hierarchic marking menus. In CHI '93: Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, 482--487.

[Mealy07] Mealy, E., Carrington, D., Strooper, P., and Wyeth, P. 2007. Improving usability of software refactoring tools. In ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference. IEEE Computer Society, Washington, DC, USA, 307--318.

[Murphy06] Murphy, G. C., Kersten, M., and Findlater, L. 2006. How are Java software developers using the Eclipse IDE? IEEE Software. 23, 4, 76--83.

[Opdyke90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In SOOPPA '90: Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications, September 1990.

[Roberts99] Roberts, D. B. 1999. Practical analysis for refactoring. Ph.D. thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. Adviser-Ralph Johnson.

[Squeak] Squeak. 2008. The Squeak Foundation. Computer Program, http://www.squeak.org/

[Weißgerber06] Weißgerber, P. and Diehl, S. 2006. Are refactorings less error-prone than other changes?. In Proc. of the 2006 International Workshop on Mining Software Repositories (Shanghai, China, May 22--23, 2006). MSR '06. ACM Press, New York, NY, 112--118.

[Vstudio] Visual Studio. 2008. Microsoft Corporation. Computer Program, http://msdn.microsoft.com/vstudio/.

[Xcode] Xcode. 2008. Apple Inc. Computer Program, http://developer.apple.com/tools/xcode/.